Southern New Hampshire University

CS 300 Pseudocode Document

Example Function Signatures

Data structures:

```
struct Course
      string courseId
      string name
      vector<string> prerequisites

vector<Course> courses;

class HashTable
struct HashNode
Course course
            unsigned int key
            HashNode *next
      vector<HashNode> nodes
      unsigned int tableSize

class BinarySearchTree
      TreeNode root

struct TreeNode
      Course course
      TreeNode left
      TreeNode right
```

Vector Algorithms:

```
void loadVectorCourses(string filename){
      open filename
      if file is not open
            print "error"
            exit

      string line

      while not end of file

      for each line in file
            vector<string> courseInfo
            split line into separate strings using , to seperate
            each string saved into course info
                  if < 2 strings on a line
                  print error, file format incorrect
                        exit
                  else
                  create course with Course struct
                  save courseInfo at zero to courseID
```

```
                  save courseInfo at one to name
                  for loop saving each prereq into prereq vector
                  insert course into vector at i
      close file
}

void searchCourse(Vector<Course>& courses, String courseNumber) {
      for all courses
            if the course is the same as courseNumber
                  print out the course information
                  for each prerequisite of the course
                        print the prerequisite course information
}


void printCourses(vector<Course>& courses, int begin, int end){
sort(courses)
for all courses
print out the course information
                  courseID, course name, prerequisites if any
}

void sort(vector<Course>& courses) {
      set low to begin
      set high to end
      if low courseID greater than or equal to high courseID
            return

      set lowEndIndex to partition(bids, low, high)

      call quickSort(bids, low, lowEndIndex)

      call quickSort(bids lowEndIndex + 1, high)

}

void parition(vector<Course>& courses, int begin, int end) {
      set low to begin

      set high to end

      set mid to low + (high - low)/2

      set pivot to title at bids[mid]

      set done to false

while !done

{
```

```
      while title at bids[low] < pivot

            ++ low

      while pivot < title at bids[high]

            --high

      if low greater than or equal to high

                  set done to true

            else

            {

            swap low and high titles

            ++low --high

            }

      }

return high

}

}
```

Hash Table Algorithms:

```
void loadCourses(string filename){
      open filename
      if file is not open
            print "error"
            exit

      string line

      while not end of file

      for each line in file
            vector<string> courseInfo
            split line into separate strings using , to seperate
            each string saved into course info
                        if < 2 strings on a line
                              print error, file format incorrect
                              exit

                        else
```

```
                              create course with Course struct
                              save courseInfo at zero to courseID
                              save courseInfo at one to name
                              for loop saving each prereq into prereq
                              vector
                              insert course into HashTable at key =
                              courseId
}
void insertCourses(HashTable<course> courses, Course course){
    convert courseId to string and hash it to create key

    retrieve whatever node is at the key, store in cur

    if there is no entry found at the key

    create a new node with the bid that was passed

    insert it into the key

    else

        if a node exists but has the default key value

    overwrite the default values with the new information

    else (collision)

        iterate through chain to find next open node

        set next value of last node in chain to the new node

}
unsigned int Hash(HashTable<Course> courses, int key) {
return key % tableSize

}
void PrintAll() {
sort by courseID
    loop from nodes begin to nodes end
        if key does not equal default value
            print courseID, course name, and prerequisites
        set node to next node
            iterate through chain
            print courseID, course name, and prerequisites
}

Course searchCourse(HashTable<Course> courses, String courseID) {
    create key out of courseID
    if entry found at key
    return the course
    if no entry found at key
    return empty course
    iterate through chain
```

```
      if current node course matches
      return course
}

void printCourse(HashTable<Course> courses, String courseID) {
Course found = searchCourse(courses, courseID)
      print courseID, course name
      for loop to print prerequisites
}
```

Binary Search Tree Algorithms:

```
BinarySearchTree loadCourses(string filename){
      open filename
      create binary search tree
      if file is not open
            print "error"
            exit

      string line

      while not end of file

      for each line in file
            vector<string> courseInfo
            split line into separate strings using , to seperate
            each string saved into course info
                        if < 2 strings on a line
                              print error, file format incorrect
                              exit

                        else
                              create course with Course struct
                              save courseInfo at zero to courseID
                              save courseInfo at one to name
                              for loop saving each prereq into prereq
                              vector
                              insert course into Tree
return tree
}

void insertCourses(BinarySearchTree<course> courses, Course course){
if tree is empty
            set root to new node(course)
      else
            call add node pass root and course

}
void addCourse(Node node, Course course){
```

```
        if node courseNumber is larger than course courseNumber add
    new course to left

            if no node in left add there

            else recursively go down the left side

        else if node courseNumber is smaller than course
    courseNuumber add     new course to right

            if no node in right add there

            else recursively go down the right


}
void printCourse(BinarySearchTree<Course> courses, String courseID) {
Course found = searchCourse(courses, courseID)
    print courseID, course name
    for loop to print prerequisites
}

Course searchCourse(Tree<Course> courses, String courseNumber) {
    iterate through tree
        if courseNumber matches return matched course
        else if traverse right if courseNumber is larger than
    curNode
        else traverse left as it is smaller
    return empty course if not found

}

void PrintInOrder(Node node) {
if node not null

        recursive call to left

        display node info

        recursive call to right

}
```

Main Menu:

```
print Main Menu
print 1. Load Courses
print 2. Display Ordered List of Courses
print 3. Display Specific Course
print 9. Exit
```

```
switch statement with user input
           case 1
           loadCourses(filename)
      case 2
      if vector
                      printCourses(courses, courseNumber)
                 if hash table
                      printAll()
                 if BST
                      PrintInOrder(root)

      case 3
      if vector
      searchCourse(courses, courseID)
      else
                 printCourse(courses, courseID)
      case 9
      exit
```

Example Runtime Analysis

Vector:

Code	Line Cost	# Times Executes	Total Cost
`open file`	1	1	1
`if file not open error`	1	1	1
`string line`	1	1	1
`while not end of file`	1	n	n
`for each line in file`	1	n	n
`split line by ','`	c	n	cn
`if <2 strings print error`	1	n	n
`create course w id and name`	1	n	n
`iterate through preq and add to course`	c (constant less than three)	n	cn

insert course into vector at i	1	n	n
		Total Cost	5n + 2cn +3
		Runtime	O(n)

Benefits + Cons of the Vector:
The vector may be the simplest to implement, but it is the least efficient of the two data structures. The vector is best at direct access, with O(1) through using the .at() function, but it falls short when searching and sorting in comparison to hash tables and binary search trees. Here, a quicksort algorithm is implemented which is normally (n log n), but at its worst can be n^2. The search is equally poor, with a O(n). In this case, c stands for a constant, as we guaranteed will have a small number of constants for both number of lines of courses and prerequisites.

Hash Table:

Code	Line Cost	# Times Executes	Total Cost
`open file`	1	1	1
`if file not open error`	1	1	1
`string line`	1	1	1
`while not end of file`	1	n	n
`for each line in file`	1	n	n
`split line by ','`	c	n	cn
`if <2 strings print error`	1	n	n
`create course w id and name`	1	n	n
`iterate through preq and add to course`	c (constant less than three)	n	cn
`insert course into hash table`	1	n	n
		Total Cost	5n + 2cn +3
		Runtime	O(n)

Benefits + Cons of the Hash Table:

The hash table is more of an intermediate data structure. It's efficient for searching and inserting, which is great when it comes to loading in the courses. Our hash table uses chaining to deal with collisions, which can also make it inefficient due to iteration through linked lists. However, it shouldn't present a huge issue as we have a minimal amount of courses to choose from. In this case, c stands for a constant, as we guaranteed will have a small number of constants for both amount of lines of courses and prerequisites.

Binary Search Tree

Code	Line Cost	# Times Executes	Total Cost
`open file`	1	1	1
`if file not open error`	1	1	1
`string line`	1	1	1
`while not end of file`	1	n	n
`for each line in file`	1	n	n
`split line by ','`	c	n	cn
`if <2 strings print error`	1	n	n
`create course w id and name`	1	n	n
`iterate through preq and add to course`	c (constant less than three)	n	cn
`insert into BST`	log n	n	n log n
		Total Cost	4n + 2cn +n log n +3
		Runtime	O(nlogn)

Benefits + Cons of the Binary Search Tree:

The binary search tree always remains in order, so a sort is unnecessary. This makes displaying the list of courses (option 2) in order extremely simple. It also allows for search time to be cut down due to the sorted structure of the tree. Inserting takes as much time as searching, and if the tree becomes heavily imbalanced, it's efficiency can decrease. Because our csv file is out of order, this shouldn't be a problem, and the tree should be fairly balanced. In this case, c stands for a constant, as we guaranteed will have a small number of constants for both number of lines of courses and prerequisites.

Which I Will Choose for my Code:

The binary search tree is the most efficient data structure and would be best to implement for my code. Although the binary tree may appear less efficient when it comes to loading, it makes up for it in its ordered structure. This ordered structure will help with insertion and searching. It also helps because there is no need to sort before displaying, as it can be easily displayed using the in-order function. I believe having a slightly longer load time is okay in this case.